4. MASM$PF1 through MASM$PFn, up to MASM's configured maximum (more @USE names)
5. The source input file, if any,
6. The source output file, if any,
7. The relocatable output file,
8. PROC$ or SYSLIB$*PROC$,
9. MASM or SYSLIB*MASM,
10. SYSLIB or SYSLIB*SYSLIB,
11. RLIB or SYS$*RLIB$.

Check your MASM reference for the exact search order and related considerations.

If MASM can not find the proc label defined in any of these files, it will flag the referencing line with an I flag and create a NOP instruction. The M option on the MASM processor call changes the search order somewhat, for special purposes. (By the way, the search order definition in the MASM manual is nowhere near the $PROC directive description; it is in section 2.4. Heads up!)

What does the search order imply for most uses?

- There are system-defined procs in SYSLIB*SYSLIB and SYS$*RLIB$, and you can't change those.
- For small applications, you can just put the proc in the same file as the source input or relocatable output.
- For much larger applications, it's a good idea to put the proc symbolics into the same file as the MASM symbolics and @PDP them into a dedicated proc file. An @USE ASM$PF,<file> makes the proc file known to MASM at assembly time.

If a proc seems to be misbehaving, make sure you have recently @PDPed it. Check the date and time on the proc element. The safest way to ensure an up-to-date proc element is to @PDP it in the same runstream that assembles the MASM which uses the proc. Or, if the proc is quite stable and you want a minimum of overhead, just @PDP manually it after each change. The author prefers never to type in any piece of ECL more than once, and to reduce the risk of error, puts the @PDP right in the assembly runstream.

16.6 The DEF Directive

If all a proc is going to do is define labels with $EQUs and $EQUFs, PDP and MASM provide a shortcut, the DEF directive. It tells MASM to put a bunch of labels into its dictionary. Here is a sample use of DEF:

```
            DEF
RTLGRN      $EQUF       0,X6,H1     . GRANULES REQUESTED
RTLFLG      $EQUF       0,X6,S4     . FLAG: @RTL OR LIJ
RTLPOS      $EQUF       0,X6,S5     . FLAG: POSITIONS OR TRACKS
RTLFUN      $EQUF       0,X6,S6     . FUNCTION FIELD
RTLFNM      $EQUF       1,X6        . TWO-WORD FILE NAME
RTLNIT      $EQU        0           . INITIALIZE FUNCTION
RTLSNP      $EQU        1           . SNAP BANK FUNCTION
RTLDEF*     $PROC       0           .
            END
```

DEF is a shortcut for MASM, not for you; it reduces the overhead required for MASM to add the labels to its dictionary. Note also that the labels defined by DEF do not need asterisks to externalize them.

You may only use DEF in procs defined using PDP. (The MASM $DEF directive has a slightly different use, beyond the scope of this book.)

16.7 Changing Location Counters

So far this chapter has referred to "the current value of the current location counter". The examples have also assumed that the code generated by a proc will go into the same location counter as the surrounding code. But a proc made for use among instructions might need to generate data, or vice versa. Or, a data-generating proc might need to switch between even location counters. A post mortem dump can be easier to read if different types of data are separated into different location counters:

- Literals,
- Single-cell data areas and edit packets,
- Print lines, print packets and assign images,
- Large data buffers,
- Input/output packets, and
- Any critical data you must find quickly in a PMD.

MASM provides a third subfield on the $PROC directive's operand field, which tells which location counter to put the output of the proc into. If it is omitted, MASM uses the current location counter.

If you do separate data types like the list above, you will have to choose your strategy from among these:

1. Each proc simply generates data under the current location counter.

2. The location counter number is specified on the $PROC directive, letting MASM switch counters for the duration of the subassembly.

3. Each proc changes to pre-established location counters for the data types it generates, assuming it is surrounded by other calls on itself.

4. Each proc is coded without knowledge of the current location counter, switching location

counters only for the data it generates. Each proc is responsible for saving and restoring the current location counter.

Every proc example shown so far uses the first strategy; they simply use the current location counter.

Here is an example of the second strategy:

```
P           $PROC,,2
ERRMSG*     $NAME
Q           P(1,1)
*           PF          2,$-Q,Q
            $END
```

This proc generates all of its output under location counter 2. The current location counter is unaffected.

Here is an example of the third strategy:

```
P           $PROC
ERRMSG*     $NAME
$(2)        .
Q           P(1,1)
$(0)        .
*           PF          2,$-Q,Q
            $END
```

In other words, this proc expects the current location counter to be 0. It puts the print image in location counter 2, then switches back to location counter 0 to generate the print packet.

The last strategy, saving and restoring location counter numbers, takes a little more work. MASM defines a label, $ILCN, as the Initial Location Counter Number. This is the number of the location counter which was active on entry to a proc. Taking the previous example, let's assume that we want print images to be generated under location counter 4, but the packets generated under the current location counter. Here's how the example would look for that requirement:

```
P           $PROC
ERRMSG*     $NAME
$(4)        .
Q           P(1,1)
$($ILCN)    .
*           PF          2,$-Q,Q
            $END
```

MASM remembers the current location counter on entry to the proc, then the proc returns to that by using $($ILCN).

If you do choose to change odd location counters, don't outsmart yourself. By definition, switching from $(1) to $(3) creates 2 non-contiguous areas. If execution is proceeding in code under $(1), and it gets to the end of what you have defined as $(1), the CPU won't know it. The CPU will keep right on executing instructions until one generates a guard mode, an illegal operation, or whatever. If you switch location counters, be sure to switch back, or make sure the code at the end of a location counter

has some sort of exit: a jump, an exit ER, or an LMJ from which there will be no return.

16.8 How To Write a Proc

As soon as you start writing a similar group of lines more than once, think "am I going to write this sequence again?". If the answer is yes, think about using a proc to generate that sequence.

- Write out a sample of the sequence of lines that the proc will generate. Underline or highlight the sections that will vary according to parameters which may change for each use of the proc. (Hint: these will become parameters on the proc references.)

- Write a sample proc reference. As a rule, it is best to put parameters which will not repeat, closer to the proc (or entry point) name. Repeated fields should be last; non-repeated fields will be easier to identify within the proc. Put items which are closely associated together in fields by themselves, especially if they will be repeated. (An example of subfields which should be paired alone in fields are test values and corresponding jump addresses.)

- Code the proc itself. If necessary, write field and subfield numbers above your sample proc reference to aid in writing the proc.

- After you have assembled your code, check the instructions or data generated by the proc. This is one of those times that some familiarity with instruction op codes and partial word modifier values will help. At the very least, remember that a j field of 01 is H2, and 02 is H1.

16.9 Listing Control

Occasionally, your code may generate either a volume or type of data that you don't want to see listed. For example, here is a proc to fill a buffer with zeroes:

```
P           $PROC                .
ZRES*       $NAME                .
            $REPEAT     P(1,1)   .
            +           0        .
            $ENDR                .
            $END                 .
```

(The $REPEAT/$ENDR directive pair will be explained later in detail. MASM repetitively assembles the lines within the scope of the pair.) Using the proc is simple:

```
BUFFER      ZRES        100      .
```

What do you suppose the listing will look like if you code this:

```
BUFF2       ZRES        1000     .
```

You will get about 20 pages of nothing but zeroes, that's what. A $UNLIST directive will turn off that listing, and a $LIST will turn it back on. What follows is an improved version of the example:

```
P           $PROC
ZRES*       $NAME
            +           0
            $UNLIST
            $REPEAT     P(1,1)-1
            +           0
            $ENDR
            $LIST
            $END
```

You are probably wondering why one word of zeroes is generated before the $REPEAT directive, outside the $UNLIST/$LIST pair. If the $UNLIST is before the first line of code generated by the proc, the $UNLIST makes the proc reference line disappear (not listed). Generating one word of zeroes before the $UNLIST allows the proc reference line to be listed.

If you use $UNLIST in the main assembly, it turns off listing of the assembly until the next $LIST directive. The $UNLIST itself is not listed, but the $LIST directive is listed. The author prefers to see even proc'd-off code listed in the main assembly, if only for documentation purposes, using the $UNLIST only within procs.

16.10 The $GO Directive

You can "spice up" your procs with the $GO directive. It causes MASM to stop interpreting lines within a proc, and begin interpreting at a new place. The place it begins interpreting again must be labeled with a $NAME directive. The label on the $NAME directive need not be externalized, so you can prevent conflicts with other names in the program.

You can use $GO with a conditional directive like $IF or $ON to change the code generated by a proc according to the parameters presented. This modification of the ZRES proc uses $GO to end the proc definition when the parameter is only 1. (Strictly speaking, it is not really necessary, because the $REPEAT will do nothing if the repeat count is zero.)

```
P           $PROC
ZRES*       $NAME
            +           0
            $IF         P(1,1)=1
            $GO         ZEND
            $ENDF
            $UNLIST
            $REPEAT     P(1,1)-1
            +           0
            $ENDR
            $LIST
ZEND        $NAME
            $END
```

16.11 Procs and Subroutines

Just in case there may be some lingering uncertainty: Procs and subroutines are not equivalent. A

proc defines a variable sequence of lines which may be invoked in line at assembly time, most often generating instructions or data. A subroutine is a subprogram which is invoked by an instruction at execution time to perform a sequence of operations. A subroutine is present at only one place at execution time, where the code generated by a proc may be present at a number of places, with variations according to the parameters presented to the proc. A subroutine may be assembled separately and included at collection time; a proc has meaning only at assembly time.

16.12 Some Handy Procs

Once in a while, you may find yourself writing code that must use the ASCII character set, but you have to code an in-line snap. Here is a proc to do the job:

```
P           $PROC       .
M$SNAP*     $NAME       .
            L$SNAP      $CFS(P(1,1)),P(1,2),P(1,3),P(1,4)  .
            $END        .
```

The $CFS converts the snap identifier from the prevailing ASCII to Fieldata. The other parameters from a reference to M$SNAP are passed on to L$SNAP. Use M$SNAP exactly the same as L$SNAP, except it can be used while a $ASCII directive is in effect:

```
            M$SNAP      'SNAPID',7,2,TABLE5  .
```

Another variation will allocate the snap packet under a different location counter:

```
P           $PROC                       .
D$SNAP*     $NAME                       .
$(0)                                    . PACKET UNDER $(0)
SNPAKT                                  .
            $GEN        P(1,1)          . SNAP ID W/LOCAL LABEL
            $GFORM      3,P(1,2),15,P(1,3),18,P(1,4) . XAR, ETC.
            +           0               . A0 SAVE HERE
$($ILCN)    .                           . BACK TO ORIGINAL L.C.
            S           A0,SNPAKT+2     . SAVE A0 OVER ER
            L,U         A0,SNPAKT       . PACKET ADDRESS
            ER          SNAP$           .
            $END                        .
```

Use D$SNAP exactly as you would L$SNAP; D$SNAP expects a $FDATA directive to be in effect. The $GFORM directive creates a SNAP$ parameter word of three fields, respectively 3, 15, and 18 bits wide. The $($ILCN) switches back to the location counter in effect at the point that the proc was referenced.

Since SNPAKT is not externalized, it is local to the proc. No D flags will result if the proc is used multiple times.

One last proc will save time testing for multiple values of a single field (variable). Several modern languages provide some sort of "CASE OF" construct. What follows is one for MASM:

```
P           $PROC           .
CASEOF*     $NAME           .
*           L           A15,P(1,1),P(1,2) . GET THING TO TEST
I           $REPEAT     P-2   . LOOP ON VALUE/TAG PAIRS
            TNE,U       A15,P(I+1,1)  . TEST A VALUE
            J           P(I+1,2),P(I+1,3)  . JUMP ON IT
            $ENDR             . END OF $REPEAT
            $END              . END OF PROC
```

Here is the general form for using CASEOF:

```
            CASEOF      <field> <value>,<tag> <value,<tag> ...
```

In other words, give the proc a field to test, followed by pairs of values and places to jump to if the field matches the corresponding values. The jump-to tag may include an index register. An example should make it clearer:

```
            CASEOF      IOSTAT,A0  0,DISK05  1,DISK25  040,0,X11
```

This proc reference will generate the code to test the field IOSTAT indexed by A0 (note the P(1,1),P(1,2) on the load instruction). If IOSTAT's value is 0, the code will jump to DISK05; if 1, to DISK25; and if 27, to 0 indexed by X11. If there are more subfields than will fit on a single source line, you can continue the proc reference on as many lines as needed:

```
            CASEOF      IOSTAT,A0  0,DISK05   ; OK STATUS
                                   1,DISK25   ; END OF TAPE
                                   040,0,X11  . STILL NOT DONE
```

You can even put comments after the semicolons to label the conditions each test value represents. It is a good idea to put the test values in order by the likelihood they will be encountered. That is, put the most likely value first, the next most likely, and so on. That way, execution time will be minimized; every test will be executed until the machine finds a match.

You can apply the principles embodied in all these example procs to make your own. Copy these procs to your own programs; study them and understand how they work.

Exercises

E16-1. Write a single proc with multiple entry points which will generate an ER to exit. The ER generated by the proc must be dependent on the name used to enter the proc. The $NAMEs and corresponding ERs should be 'QUIT' (EXIT$), 'ERROR', (ERR$), 'MABORT' (EABT$), and 'ABORT' (ABORT$). Make sure that if there is a label on the proc reference, it will be given the value of the current location counter. (Hint: use a waiting label.)

E16-2. Here is a proc reference:

```
        CRUNCH,12   ABC,DEF   QZR,99,6,42,HIKE
```

Write the parameter reference forms for the following items in the proc reference. (You may assume that the proc has the unexternalized name P, so you can write P(-,-):

```
ABC
42
QXR
12
HIKE
```

Chapter 17: MASM Operators and Expressions

17.1 Introduction

MASM provides the ability to combine constants and variables in formulas that, in their simplest form, are reminiscent of Fortran. But, because MASM has more data types than Fortran, there are more operators available, and the possibility of more complex formulas. Like Fortran, MASM operators have a strict order of precedence, and that order is well documented. You can refer to the precedence chart in your MASM manual, section 4.4.2.1. This chapter will begin with the simplest, most familiar operators, and proceed into MASM-only territory.

17.2 Arithmetic Operators

These operators include the usual + and - for addition and subtraction, and * and / for multiplication and division. While MASM supports floating point arithmetic within an assembly, such use is rare. In integer arithmetic, of course, no fractional part is maintained for a number. Addition, subtraction, and multiplication are exact. MASM uses 72 bits plus a sign for internal arithmetic, trimming to narrower fields only when necessary, so you rarely need to worry about overflow or loss of precision.

Integer division, just like in Fortran or Cobol, discards the remainder. So, dividing 14 by 3 gives a quotient of 4. If you need to obtain the remainder of a division, there is the /// operator. The expression 14///3 is 2 (4 times 3 is 12, from 14 is 2).

One other operator that may be new to you is the covered quotient, or // operator. It divides the first quantity by the second; if there is a remainder, the quotient is bumped by 1. The quantity 14//3 is 5. A real-world application of this operator might help make the covered quotient concept clearer: Let's say you need to box up a number of auto parts, and only 5 fit in a box. How many boxes will you need for 26 parts? There will be 5 full boxes, and 1 with only 1 part, for a total of 6 boxes.

Let's say you are allocating buffer space for an ER which can return a variable amount of data. If it returns 5 words per disk type, plus 10 words per disk unit, how many words must you reserve? The answer is obviously based on an estimate of the number of disk types and disk units. (Your estimate must almost always be on the high side.) Set both of these up as parameters, and base the $RES directive on those two variables:

```
DTYPES    $EQU    2              . MAX DISK TYPES EXPECTED
DUNITS    $EQU    32             . MAX DISK UNITS EXPECTED
ERBUFF    $RES    DTYPES*6+DUNITS*10 . BUFFER SPACE
```

STYLE: You must parameterize your assumptions in this manner if you want your code to be maintainable. The $EQUs for DTYPES and DUNITS are good candidates for a "configuration" proc, especially if the program consists of multiple assembly elements. At the very least, these $EQUs should be placed at beginning of your MASM element, and identified there as configuration-dependent variables. Wrong answers or odd error terminations can result from exceeding parameters like these. Make sure your assumptions are well-documented.

Why don't we need to put the subexpressions DTYPES*5 and DUNITS*10 in parentheses to get the right answer? The order of precedence dictates that the multiplications are done before the addition. (Check the precedence table in the MASM manual.) If you are ever in doubt (hopefully with more complicated expressions than this), you can use parentheses to force the expression to be evaluated in the order you need.

The last two arithmetic operators are unary + and -. They create the arithmetic positive and negative, respectively. The unary - can also be used to create a logical negative, inverting each bit setting, 0 to 1 and 1 to 0.

17.3 Logical Operators

We have the mathematician George Boole to thank for logical variables. MASM operates on logical variables in a bit-for-bit fashion. You can effect simple yes-or-no decisions, operating on single-bit values, or you can operate on 36- or 72-bit values. MASM does not distinguish between them. You will use these operators mots often combining bit settings for:

- Sections of code to be assembled conditionally, based on configuration variables. One example is debugging code.
- Bit masks to test option sets. If your program will do different jobs based on execution options (@XQT,<opt> or @PROG,<opt>), some combinations may not make sense together.
- Bit masks to test statuses returned from ERs or subroutine calls.

The first logical operator is ++, for logical OR, also called logical addition. For each bit position combined using ++, logical OR returns a 1 if either operand is set. Here is a truth table for logical OR. It shows the result for each possible combination of operands:

		Operand 2	
	++	0	1
Operand 1	0	0	1
	1	1	1

If this table looks a lot like the logical OR table in Chapter 11, that's no accident; they are identical.

What is the result if we logically OR 5 and 3? Express it as

```
RESULT      $EQU        5++3
```

Break down the operands into binary, and see. Five is 1012, and 3 is 0112; 1 OR 0 is 1, 0 OR 1 is 1, and 1 OR 1 is 1. The result is 111_2, or 7.

Logical AND (**), also called logical multiplication, uses this truth table:

		Operand 2	
	**	0	1
0	0	0	0
p			
1	1	0	1

In other words, the result is 1 only if both operands are 1. What happens if we logically AND 5 and 3, like this:

```
RESULT      $EQU        5**3        .
```

Again, 5 is 1012, and 3 is 011_2. One AND 0 is 0, 0 AND 1 is 0, and 1 AND 1 is 1. The result is 001_2, or 1.

The last logical operator is —, logical subtraction or exclusive OR (XOR). It uses this truth table:

		Operand 2	
	--	0	1
0	0	0	1
p			
1	1	1	0

In other words, the result is a 1 bit if only one of the operands is a 1. What is the result of logically subtracting 5 and 3? (Order of the operands does not matter.)

```
RESULT      $EQU        5-3         .
```

The answer is: 1 XOR 0 is 1, 0 XOR 1 is 1, and 1 XOR 1 is 0; the result is 1102, or 6.

Suppose we combine all three in a single expression? How does the order of precedence affect the result?

```
RESULT      $EQU        4**5++3-8   .
```

Precedence calls for the ** to be done first, then the ++ and --. The operators ++ and -- enjoy the same precedence, and ** has the next higher precedence. RESULT is 15; here is how you get it:

```
4       0100
5       0101
        ____  **  (AND)
        0100
3       0011
        ____  ++  (OR)
        0111
8       1000
        ____  --  (XOR)
        1111
```

The NOT operator, \ (backslash), operates on only one operand, logically negating it. The \ operator differs from the other logical operators; its result is not a bitwise negation. If the operand is nonzero, the result is zero; if the operand is zero, the result is 1. Here is an example use of \:

```
DEBUG       $EQU        1           . TESTING ONLY
LIVE        $EQU        \DEBUG      . NON-DEBUG FLAG
```

One last operator, */, shifts the first operand by a number of bits specified by the second operand. If the second operand is positive, the shift is to the left; if negative, to the right. For example, 1*/1 is 2, 3*/4 is 060, and 017*/-2 is 3. Bits shifted off the right with a negative shift count are lost. Note that shifting an operand is like multiplying or dividing it by a power of 2.

17.4 String Operators

The string concatenation operator : attaches one string to another, as in this example:

```
FNAME       $EQU        'JOHN'      . FIRST NAME
LNAME       $EQU        'DOE'. LAST NAME
FULNAM      FNAME:' ':LNAME .
```

What will be the result in memory at the tag FULNAM? It will be the concatenation of JOHN, a space, and DOE, the string 'JOHN DOE'.

Two more string operators control the way strings are laid into memory. A postfix L stores a string as left justified, space filled. The Fieldata and ASCII edit packages in SYSLIB require you to use this for strings passed to the 'message' procs (E$MSG and A$EMSG).

The postfix R stores a string as right justified, zero filled.

Here is an example use of the postfix L:

```
PF          $FORM       12,6,18     .
            L           A0,(PF 1,2,('ERR EXIT'L)) . PRINT PARAM
            ER          PRINT$              .
```

The outer literal creates a PRINT$ parameter word, while the inner literal puts the string 'ERR EXIT' into two more words, left justified, space filled. Can you see which word(s) will be put into memory first, the print parameter word, or the string? Hint: MASM suspends assembly of the PF while it assembles the character string. Therefore, the PF word will be after the character string in the literal pool. Intuitively speaking, MASM cannot assign the address to put into the PF until it has stored the character string.

17.5 Relational Operators

The relational operators return binary results. For example, the result of 1>5 is 0, or false; 5<=5 is 1, or true. Here are the relational operators:

- < is less than,

- <= is less than or equal to,
- = is equal,
- > is greater than,
- >= is greater than or equal to,
- = is equal, and
- <> is not equal.

You can combine these operators in expressions with the logical operators. For example:

```
DEBUG       $EQU        1               . GLOBAL DEBUG CONTROL
TRACEL      $EQU        2               . TRACE LEVEL
DOTR1       $EQU        (TRACEL>=1)**DEBUG  . TRACE TYPE 1
DOTR2       $EQU        (TRACEL>=2)**DEBUG  . TRACE TYPE 2
```

The trace flags DOTR1 and DOTR2 are turned on (set to 1) only if DEBUG is turned on, and the trace level is at least 1 for DOTR1 and 2 for DOTR2. You can use expressions like these for conditional assembly controls. The = operator can actually be used to negate a binary variable like DEBUG in the code fragment above:

```
LIVE        $EQU        DEBUG=0     . LIVE MODE
```

LIVE is set (1) only if DEBUG is zero.

17.6 Single and Double Precision Control

Most MASM programming uses single precision fixed-point arithmetic. That is, while MASM maintains 72 bits of precision internally, it creates 36-bit words by default. You may have occasion to expand a number into a two-word field. Use a postfix D to create a 72-bit, 2-word field:

```
DWFLD       +       7D          . DOUBLE-WORD 7.
```

MASM ignores the magnitude of the number to create the two words. The result in DWFLD will be 000000000000,000000000007. (The 7 is right-justified.)

MASM maintains a precision attribute with each value, and it is normally set to single. If you have created a value with the double precision attribute, you can override that to create a single-word field with the S postfix operator:

```
SWFLD       +       (7D*57)S    . SINGLE-WORD 399.
```

17.7 Conditional Expressions

Conditional expressions are handy for compressing decisions on multiple values into a single line. They also maintain control information, allowing you to choose one of two instruction mnemonics based on a variable. The if-then construct is:

```
a->b     or

a->b!c
```

In other words, if condition a, then the value b. The second form allows for an alternative: if condition a, then value b, else value c. The condition a is also any variable. If its value is non-zero, it is logically true. If zero, it is logically false. Here is an example use of a conditional expression:

```
DEBUG       $EQU        1              . DEBUG CODE CONTROL
TABLE3      +           DEBUG->7!0,TABLE4  .
```

H1 of TABLE3 is 7 if DEBUG is set, else it is 0. H2 is the value TABLE4, regardless of the value of DEBUG. If-then and else operators can be nested. Check your MASM manual for interpretation of nested if-thens.

Exercises

E17-1. Recall the example of the covered quotient: Five auto parts fit in one box. Code the expression to calculate the number of boxes needed for 84 parts.

E17-2. If 2 of the boxes fit in a crate, code the expression for the number of crates needed for 57 parts. Evaluate the expression.

E17-3. Is there a second way to code the expression in E17-3? If there is, does it come up with the same answer?

E17-4. Evaluate the following expressions, being careful to observe precedence. (Hint: If you are not sure of your answers, try assembling the expressions.)

```
2+17**3++27
100*/6
100-200
\(5-5=0)
```

E17-5. Code string $EQUs for each of your names (first, middle, and last); concatenate them into contiguous memory locations.

Chapter 18: More MASM Directives

18.1 $GFORM - Generalized $FORM Directive

Chapter 9 described the $FORM directive, which is used to fill a word or two with data fields on arbitrary boundaries. $FORM associates a name with a word format, and the name can be used over and over in the operation field of a MASM line. $GFORM divides a word (or two) on boundaries that are just as arbitrary. Each field value is paired with a bit width, but the word division applies for only one line. Here is the format of a $GFORM line:

label $GFORM $width_1, value_1, width_2, value_2, \ldots, width_n, value_n$

$value_1$ is placed in a field $width_1$ bits wide, $value_2$ in a field $width_2$ bits wide, and so forth. The sum of $value_1$ through $value_n$ must be either 36 or 72. The label on the line is not associated with any form; it is given the value of the current location counter.

18.2 $DO - Repeat a Single Line

The $DO directive assembles a single line multiple times, using an assembly-time loop counter. You can reference the counter by putting a label on the line containing the $DO. (Note that the label is not given the current value of the current location counter.) Here is the format of a $DO directive:

<label> $DO <repeat> ,<line> .

There are three possible subfields within <repeat>: <start>, <end> and <step>.

- If <repeat> has only one subfield (like the value 17), then that expression is assumed to be <end>; <start> and <step> are each 1.
- If <repeat> has two subfields (like 2,20), they are assumed to be <start> and <end>; <step> will be 1.
- If <repeat> has three subfields, they will be the starting value of the counter, the final value of the counter, and the increment for each iteration.

You must put at least one space between the last subfield of <repeat> and the comma before <line>.

The <label> is assigned successive values of the loop counter. You can reference <label> within <line>, but the value of <label> is meaningless outside the scope of <line>.

The <line> may optionally have a label beginning immediately after the comma. If you want to omit the label, one or more spaces between the comma and the next non-blank character will do the job.

The $DO directive is a lot easier to use than describe. Here is an example using $DO to create 20 words of data. H1 of each word will contain the word number, counting up from 1 to 20, and H2 will count down from 20 to 1:

```
TAB84.
ICNT        $DO         20    ,   +   ICNT,21-ICNT .
```

The label TAB84 is given the location of this table on the line before the $DO directive. Since there are no standard tab settings for anything between the operation field (in this case the value 20) and the comment field, the author usually double spaces between those subfields. The line assembled by the $DO itself has no label. (In fact, putting a label on the line would cause D flags.)

The line assembled by a $DO can be any valid assembly code, including data, instruction, a proc reference, or even another $DO directive.

18.3 $REPEAT - Repetitively Assemble a Range of Lines

While $DO can assemble only one line, a $REPEAT/$ENDR pair can assemble almost any number of lines. Like $DO, $REPEAT initializes a loop counter and makes it available within the scope of the $REPEAT/$ENDR pair. Here is the format of the $REPEAT and ENDR directives:

```
<label>     $REPEAT     <repeat>    .
            .
            (scope of $REPEAT/$ENDR pair)
            .
            $ENDR                   .
```

The <repeat> field has exactly the same number of subfields and interpretation as $DO; one subfield is the ending value, two are starting and ending values, and three are starting, ending, and step values. The <label> is repetitively assigned the value of the loop counter.

$REPEAT can do everything that $DO can do, and more. $REPEAT can assemble multiple lines per iteration, but $DO cannot (without referencing a proc which contains multiple lines). Here is a reprise of the $DO example, using $REPEAT:

```
TAB84.
ICNT        $REPEAT     20          .
            +           ICNT,21-ICNT .
            $ENDR .     20          .
```

The value 20 really should be made into a parameter. Here is an improved version of the example:

```
TAB84.
TABSIZ      $EQU        20          .
ICNT        $REPEAT     TABSIZ      .
            +           ICNT,TABSIZ+1-ICNT .
            $ENDR .     TABSIZ      .
```

The $ENDR shows another device for documenting what's going on in your code. The period space (". ") terminates the assembler's scan of the line. Putting the parameter for the limiting value of the $REPEAT as a comment on the $ENDR helps locate the scope of the $REPEAT/$ENDR pair, especially if they are nested inside others.

18.4 $ON and $OFF - Conditional Assembly

Chapter 17 referred to conditional assembly. You can cause code to be assembled only for debugging purposes, or to make other configuration-dependent changes at assembly time. Here is the format of a $ON/$OFF pair:

```
$ON         <switch>    .

(scope of conditional code)

$OFF                    .
```

If the value of <switch> is non-zero, the code inside the $ON/$OFF pair will be assembled. If <switch> is zero, the code inside the pair will not be assembled. $ONs and $OFFs may not be nested. Here is an example use of $ON and $OFF:

```
$ON         DEBUG       . CONDITIONAL CODE
P$RINT      MSG027      . 'I GOT HERE' MSG
$OFF .      DEBUG       . END CONDITIONAL
```

If DEBUG is nonzero, the P$RINT line will be assembled. Note that keeping track of which registers are modified by conditionally-assembled code and under what circumstances is your responsibility; don't let the registers get out of control.

You can turn on conditional code when a flag is zero, too:

```
$ON         DEBUG=0     . LIVE MODE ONLY
J           P5F010      . GO DO LIVE ER.
$OFF .      DEBUG       . END CONDITIONAL
```

Put the name of the assembly-time flag controlling the $ON/$OFF pair as a comment in column 21 of a $OFF directive to help locate the scope of the pair.

18.5 $IF, $ELSE, and $ENDF Directives

The $IF and $ENDF directives, used alone, are analogous to $ON and $OFF. (But, you must pair $ON with $OFF and $IF with $ENDF.) Use them exactly the same way. The other directives expand your control over conditional assemblies. $ELSE provides an alternative if a $IF test fails. Here is an example using $IF, $ELSE, and $ENDF:

```
$IF         DEBUG       . CONDITIONAL CODE
P$RINT      MSG027      . 'I GOT HERE' MSG
$ELSE                   .
J           P5F010      . GO DO LIVE ER.
$ENDF .     DEBUG       . END CONDITIONAL
```

If DEBUG is set (nonzero), the P$RINT will be assembled; if DEBUG is clear (zero), the jump will be assembled. The $IF/$ELSE/$ENDF set has advantages over $ONs and $OFFs:

- $IF/$ELSE/$ENDF replaces two $ON/$OFF pairs, requiring fewer lines of code, and

- The $IF/$ELSE/$ENDF set makes the "either this or that" decision precisely clear from a documentation standpoint. A reader might have to spend a few minutes comparing the $ON conditions of adjacent sections to make sure that they were complementary. (An example might be $ON DEBUG and $ON DEBUG=0.)

Can you see there is another way to do this example using $IF? You could put the "live" code first:

```
$IF         DEBUG=0     . CONDITIONAL CODE
J           P5F010      . GO DO LIVE ER.
$ELSE       .           .
P$RINT      MSG027      . 'I GOT HERE' MSG
$ENDF .     DEBUG=0     . END CONDITIONAL
```

The way you choose is left to your style. The author prefers to put a positive test first, like $IF DEBUG.

One other style point, and we'll move on. You might like to put blank comment lines before and after the $IF/$ENDF set, or even in the middle, to improve readability:

```
.
$IF         DEBUG=0     . CONDITIONAL CODE
.
J           P5F010      . GO DO LIVE ER.
.
$ELSE       .           .
.
P$RINT      MSG027      . 'I GOT HERE' MSG
.
$ENDF .     DEBUG=0     . END CONDITIONAL
.
```

The idea is not to create a double-spaced effect; the jump and the P$RINT represent larger contiguous sections of code.

18.6 $ELSF and $ANDF Directives

Think of $ELSF as a combination of $ELSE and $IF. The $ELSF directive, used within a $IF/$ENDF pair, lets you set up multiple alternatives. The multiple alternatives may be different values of the same variable, or independent tests. The last chapter used an example involving multiple trace levels. Here is one way to put them to work controlling conditionally-assembled code:

```
$IF         TRACEL=1    . TRACE LEVEL 1 ONLY
L$SNAP      ...         .
$ELSF       TRACEL=2    . TRACE LEVEL 2 ONLY
L$SNAP      ...         .
P$RINT      ...         .
$ELSF       TRACEL=3    . TRACE LEVEL 3 ONLY
... (MORE DEBUG CODE)   .
$ENDF                   .
```

Only one of the sets of lines of debug code will be assembled, depending on the value of TRACEL. You can string as many $ELSFs together between $IF and $ENDF as necessary. There need be no connection between the tests done by the $IF and the $ELSFs in a set. Bear in mind, though, that once a "true" or "set" condition is found, evaluation of the alternatives will end. This variation on the example above will not do what the creator might have had in mind:

```
          $IF       TRACEL>0    . ANY TRACE LEVEL
          L$SNAP    ...
          $ELSF     TRACEL=1    . TRACE LEVEL 1 ONLY
          L$SNAP    ...
          P$RINT    ...
          $ELSF     TRACEL=2    . TRACE LEVEL 2 ONLY
          ... (MORE DEBUG CODE) .
          $ENDF                 .
```

If TRACEL is any value greater than zero, the first L$SNAP will be assembled; the other alternatives will be ignored.

The $ANDF directive allows you to "pile up" additional tests on an $IF condition. $ANDF accomplishes the same purpose as nested $IFs, but it is more efficient and easier to read. This example shows a more likely use of increasing trace levels:

```
          $IF       TRACEL>0    . ANY TRACE LEVEL
          L$SNAP    ...
          $ANDF     TRACEL>1    . TRACE LEVEL 2+
          L$SNAP    ...
          TNZ       FLAG2       . IS FLAG 2 SET NOW?
          J         T5G050      .  NO,T5G050  YES,
          P$RINT    FLG2MG      . 'FLAG 2 IS SET AT PT 27'
T5G050    .
          $ELSF     TRACEL>2    . TRACE LEVEL 3+
.         ... (MORE DEBUG CODE) .
          $ENDF                 .
```

Once again, the tests may be independent, or they may involve the same variable. If the first $IF is not satisfied, nothing between the $IF and the $ENDF is assembled.

The $ELSE directive also can be used with $ELSFs and $ANDFs. Used with $ELSFs, it can supply an alternative set of lines which will be assembled if none of the other conditions are satisfied. Used with $ANDFs, $ELSE switches from conditionally assembling to skipping and vice versa.

Exercises

E18-1. Code something which will put 5 in an 12-bit field, GT027 in 4 bits, and 12*RBGLAS in 20 bits of a single word.

E18-2. Code a directive which will put I and I-5 into H1 and H2, respectively, of 100 words labeled TAB100. (Hint: don't put TAB100 on the $DO or $REPEAT directive line.) Start with the value 1. That is, the words should have 1 and -4, 2 and -3, 3 and -2, and so forth.

E18-3. Code a $DO which will put the following values into sequential words: 1, 7, 13, 19, and so forth, up to 25.

E18-4. Repeat E18-3, but use a $REPEAT.

E18-5. Code a jump to PH5000 which will not be assembled if DEBUG is less than 3.

Chapter 19: MASM Style

19.1 Why Develop A Style?

If you have paid any attention to the style boxes up to this point, maybe you already know the answer to the question. Just as in writing fiction, all other considerations take a back seat to clarity of expression. Saving a few hundred nanoseconds is irrelevant if it takes someone later an extra hour to understand and confirm your logic. Achieving clarity and doing a job well will put your style on the map better than anything else.. You've still got plenty of room to express your individuality.

On the other hand, learn how to save execution time using straightforward instructions. Learn the break point (or at least be aware there is one) between using a block transfer and a load/store loop to copy data. Understand the criteria for deciding between using a character vector and using hardware-specific (Extended Instruction Set) instructions to move and edit characters. (Character handling is covered in Chapter 21.)

Having reached this point in the book, you have enough knowledge to make the machine do your bidding. The remainder of the book is about tools for creating system-level applications. They will build on your knowledge of the instruction set and basic ERs to accomplish real work. While you may struggle at first to express your thought processes, keep yourself and succeeding programmers in mind. If one of two or three ways to do something are almost equally effective and efficient, choose the easiest to understand. You will benefit in several ways:

- You are less likely to make a mistake coding it in the beginning. (An application of the KISS principle — Keep It Simple, Sir.)
- You will find it easier to debug.
- Should your system configuration or some other aspect of the problem change, it will be easier to determine the boundary or limiting conditions of the program — those that make the program go belly-up.
- Someone else maintaining the program after you will have an easier time.
- You will gain a reputation as a quality programmer. Your pride and confidence will be unshakable.

The style guidelines that follow are, in some cases, a distillation of style notes elsewhere in the book. They are repeated here to expand on the original ideas. Others are not found elsewhere, because they don't logically belong anywhere in particular. Every time you feel uneasy about something in your or others' code, think about an addition to this list. How could it be done better? What more general principle can you carry to the next project?

Without further introduction, here are some style guidelines.

19.2 Build ECL For Assembly and Mapping

Before you get very far in coding your program and typing it into the machine, you may want to see what it looks like so far. That's the time to build the ECL element.

Give the ECL element a name that is consistent with other ECL you create. The author uses C/<name> for every assembly and compilation runstream he has created for years, and R/<name> for execution runstreams.

19.3 Parameterize Everything You Can

Use $EQUs for every buffer or table size which may change over time. If your initial size is a guess, make it a $EQU anyway, and document the assumptions of your guess. Put parameters which must be shared by multiple elements into a single proc element, and reference that proc in each MASM element. If both MASM and higher-level language elements such as Fortran or Cobol must share parameters, set up the parameters as Stream Generation Statements and use a small skeleton to massage them into PDP elements.

Put comments next to each $EQU explaining what the parameter is, and the restrictions which apply. Here is a very readable format to use:

```
.
DTYPES      $EQU        5             . MAX DISK EQUIPMENT TYPES
.                                     . 8481, 8480, D7053, ETC
.                                     . ADD 1 FOR EACH NEW TYPE
.
UNITS       $EQU        150           . NUMBER OF DISK HDAS
.
MSBUFF      $RES        1+DTYPES*5+UNITS*10 . MSALL$ BUFFER
.
```

(The ER MSCON$ function MSALL$ requires a constant 1 word, plus 5 words per equipment type, plus 10 words per disk unit.) The assumptions which went into these equates is easily verifiable, and a problem is easily resolved.

Make every field in a table a $EQUF. Even if you access a field in only one place, make it a $EQUF. Using $EQUFs has numerous benefits:

- You will reduce the chance of error in coding.
- You can document all the fields together, using a comment format like the above example uses.
- Even if the format of the table is cut and dried, like an I/O packet, naming the fields has the benefit of documenting which fields you plan to access.
- Rearranging a table is much easier—you just change the $EQUFs and nothing else.
- Picking off the size of an entry is easy, as in the following example:

```
TABNAM      $EQUF       0,,X5         . FILE NAME (2 WORDS)
TABGRN      $EQUF       2,H1,X5       . GRANULES
TABPOS      $EQUF       2,S4,X5       . POSITION FLAG
TABFUN      $EQUF       2,T3,X5       . FUNCTION
TABCOM      $EQUF       3,W,X5        . COM$ PACKET (3 WORDS)
TABSIZ      $EQU        (TABCOM+3)**0777777 . ENTRY SIZE
```

The last $EQU strips the partial word and index register information, giving us back the length of an entry, 6 words. If you have other code dependent on this information, it can allocate space automatically based on TABSIZ.

If you have several multi-word tables which will be one right after another, you can make that clear with another technique. Put $EQUFs for the start of each "inside" table together, then later list the $EQUFs for the individual fields of the inside tables, like this:

```
LOGBUF      $EQUF       0,X10         . LOG BUFFER - 24 WORDS
RPKT        $EQUF       LOGBUF+24     . EDIT$ PACKET - 6 WORDS
RPTBUF      $EQUF       RPKT+6        . EDIT$ BUFFER - 22 WORDS
OPBUFF      $EQUF       RPTBUFF+22    . COM$ RESPONSE - 1 WORD
OPRCOM      $EQUF       OPBUFF+1      . COM$ PACKET - 3 WORDS
DUMPKT      $EQUF       OPRCOM+3      . SNAP$ PACKET - 3 WORDS
NSPACE      $EQU        (DUMPKT+3)**0777777 . WORDS REQUIRED
...
LOGSTA      $EQUF       LOGBUF+0,,H2  . LOG$ STATUS
LOGSEN      $EQUF       LOGBUF+1,,W   . LOG$ SENTINEL
...
...
OPROCC      $EQUF       OPRCOM+1,,H1  . COM$ OUTPUT CHAR COUNT
OPROBF      $EQUF       OPRCOM+1,,H2  . COM$ OUTPUT BUFFER LOC
```

and so on. This technique allows you to move the inside tables around as necessary without adjusting the individual field $EQUFs. The index register (in this case, X10) will be carried from the first $EQUF, LOGBUF, to the other $EQUFs.

19.4 Modularize Functions

No subroutine should be bigger than you can understand easily. If the level of complexity gets away from you, the subroutine is too big. In practice, that means few subroutines should be more than 2 pages. One exception is a routine that does a number of identical, repetitive functions; it can be somewhat longer.

Using multiple layers of subroutines requires a simple, consistent method for assigning the LMJ registers. Remember that the same set of index registers, X1 through X11 and A0 through A3, have to point to tables or packets, too. The author uses the following technique, and you can adapt it for your needs:

- Reserve X11 for calls on system-defined routines (such as in SYSLIB), or as a temporary table pointer.

- Assign subroutine call registers starting with X1. Use X2 for the next level, X3 for the next, and so on down.

- Assign less-volatile table index registers beginning with X10, then X9, and so on, backwards.

- Use A0 through A3 and the "middle" X registers as volatile (changeable, short-term) table pointers. Since A0 through A3 can double as arithmetic registers, they can do more flexible jobs than X registers.

- If a routine must be called from more than one level, always call it with the index register consistent with the lowest level it is called from. In other words, if it is called both from the "1" level (which would use X2) and from the "2" level (which would use X3), always call the routine using X3.

At first glance, a program whose top level consists of 5 subroutine calls (LMJs on X1) may not seem "creative". So what? As long as you can spare X1, who cares if it changes only a few times during execution? Leave X11 calls and multitudinous return address saves to Cobol and SYSLIB. Save your decision-making power for more interesting problems than which register to call a subroutine on; this part should be a "no-brainer".

19.5 Use Registers Consistently

Use the same register and instruction sequence for the same purpose each time. In other words, if you have to calculate a table index, do the same load, shift, and add. Don't add upper from another register, multiple by a power of two, then use some other technique to accomplish the add. Using index register names on $EQUFs helps enforce consistent usage. Nobody will give your code a better "grade" for "creativity" for the number of different ways you can code the same sequence. You may actually mislead yourself, and make a new mistake for each new way you do something.

19.6 Limit Range of Unconditional Jumps

Consistent with modularizing functions is the idea that you should limit the number of jump instructions you do. Jump instructions should be either within a two-page routine or to an error path, never to return. The author has seen many MASM programs which must have been a nightmare to code, and worse to debug. JGD loops which extend over several pages should be against the law! Anything inside a loop of that size should be extracted as a subroutine by itself. Any intelligence which passes from the outer loop to the subroutine should be documented as an argument, even if it is not passed in a formal list.

Older programs also sometimes used a technique of setting an instruction address in an index register at several locations, then, several pages later, jumping on the index register. This is confusing at best, and hard to debug. It is also sloppy design; the conditions under which a routine is called must be well-defined, and the "random-jump" technique make that very hard to do.

Never code a jump to a label plus a constant (that is, TAG17+2). If more code is inserted between the label and your intended destination, the code will not work anymore, and the cause will be very hard to find. Code a label at your intended destination; make it easy on yourself.

19.7 Use Multiple Location Counters

You can use multiple location counters to separate data of different types. Put the packet for an ER call under $(0), but put the buffer into $(6) to make it easier to find in a PMD. Or, you can use PMD to dump selected location counters, perhaps even in Fieldata or ASCII format. Exclude the buffer space in the PMD if you don't want to see it.

Put literals under an odd location counters. They'll be in the I bank, where they may even end up write-protected. Since they won't change, you don't need to see them in a PMD, so you can PMD the D bank only.

Shorter PMDs mean you may be able to use a smaller, slower printer, probably closer to your desk, and possibly get your printout back more quickly.

19.8 Make Labels Into Comment Lines

A label in I bank code is important enough that it should not have to share the line with an instruction. Here is why:

- It can be the target of a jump. Under what conditions will you jump there? What will you do when you get there? Answer those questions with comments on the line. If the answer is long-winded, take several lines.
- If you use SIR (Source Input Routines) correction lines to change your source later, you will have a hard time separating the label and the instruction.

If the instruction immediately preceding the label is an unconditional jump, put an extra, blank comment line between them. That helps emphasize the break in logic at that point.

19.9 Assign Labels Systematically

Instruction labels do not have to be hard to figure out. At the time you design the program, separate it into phases, usually less than 10. The first phase begins with the label PHASE1, and the labels in that routine begin PH1. The last three characters of the names should be numbers, beginning with 010 and increasing 010 for each. If you have to insert other labels in between, make the insertions proportional (like 054 and 057 between 050 and 060). Name the next phase PHASE2, and its labels PH2nnn, and so forth.

If the program has been subjected to numerous insertions, go back and renumber the labels. Begin the renumbering at 010, increase by 010 for each, but do the editing backwards. In other words, change the last label first, then the next to last, and so forth. The idea is avoid conflicts between the old numbers and the new numbers, which are indistinguishable to the text editor.

If Phase 1 calls a subroutine, make it P1UNIT, P1SRCH, or a similar name. A routine called only by Phase 5 should be called P5WORK, and so on. Labels in these routines should be of the format PnZ000. The P is a constant, n means the phase number which uses the routine, and Z the first letter of the word describing the job it does. A routine which saves the mass storage status within Phase 4 (P4MASS) should have labels like P4M010, P4M020. A routine called from multiple phases should be called PXMISC, the X meaning any phase number.

The author has seen label conventions which included the name of every known brand of beer, constellations, and everything else, and the technique suggested above works. You might argue that P4M050 is a bit cryptic, but the format gives several clues to the context of the label. Both the label and jumps to it should be located on the same one or two pages; they won't be hard to find.

Still other label conventions require "meaningful labels", like INSERT for a section of code that inserts items in a table, or FOUND when a search locates an item. The problem with this technique is that it is hard to come up with 6 variations on INSERT or DELETE for code that does a lot of table manipulations, for example. The word FOUND does not adequately describe what was searched for. In general, there is no way that a 6- or 8-character label can substitute for a full line of clear comments on a label (see 19.11).

19.10 Unnamed Jump Targets and Coding

The next note has to do with the physical process of putting code on a page as you create a program. As you write, you may be juggling several things in your head at once: where the next jump is going, and what happens after the jump, which registers have to be left alone, and so forth. Here is a technique to make part of that easier. When you code a test followed by a forward jump (to a label not written yet), don't write the destination of the jump yet. Instead, leave the operand field of the jump blank, and put an asterisk or star in the label field, like this:

```
          TE        A5,RATPAK  . IS THIS THE PACKID?
*         J                    .  NO,          YES,
```

(This asterisk is not a waiting label, just a note to yourself.) Leave space for the jump target in the comments, too. Next, focus your attention on the code that follows the jump. As you code additional labels, continue to name them in sequence according to the rules mentioned earlier. When you get to the place in the logic which is the logical destination of the jump with no target, fill in the label. Erase the asterisk or star. The two lines above will look like this:

```
          TE        A5,RATPAK  . IS THIS THE PACKID?
          J         PH2050     .  NO,PH2050  YES,
```

As you code a long sequence, you may have several stars strung out behind you. No matter; a quick scan will tell where there are unsatisfied jumps. If you have made your comments clear and concise, you can recall what the code was supposed to do at the jump target.

This technique is superior to guessing at what the labels will be, and then having to go back and change them when an unexpected turn in the logic messes up your numbering sequence. Also, leaving jump targets unnamed until you define them helps ensure that you will define every jump target; undefined jump targets stand out clearly. You won't have to let MASM find the undefined labels for you.

19.11 Use Clear In-Line Comments

Help the reader of your code and write the intent of each line as an in-line comment. Put the period space ('. ') terminator starting in column 39, then the comment. If a sequence of loads, adds, and shifts is intended to implement a complicated formula, the comments should give a "blow-by-blow" description of the intermediate (subformula) results.

If your head is swimming as you write the code, and have to collect your thoughts, say what is in your mind in comment images. Note the assumptions you have made to that point, what you expect in the important registers, and so forth. If you are prone to confusion at that point, it is likely a later maintainer of your code will be, too. A short paragraph at the beginning of each routine is a good idea, too, looking something like this:

```
/.
.     P4MASS - SAVE MASS STORAGE STATUS.
.            DO MSALL$ AND CREATE SDF RECORD.
.
P4MASS      .
```

The /. makes MASM start a new page, and identifies this area as the start of a new subroutine. The comment lines after the /. make the subroutine name stand out.

Each test instruction should have an inline comment describing what you think you are testing. The line following a test should tell which alternative is which and where the jump will take it for that condition:

```
            TNE         A5,27       . LAST ENTRY?
            J           P4M060      .  YES,P4M060   NO
```

The first alternative (YES) is indented by one character and tells what it means if the jump is taken, followed by the jump-to label (P4M060). Two spaces later, the second alternative shows the meaning of a fall-through (NO). Meaningful in-line comments help reconstruct your intent at a later time, and help you figure out if you wrote the wrong thing to match that intent.

If the positive alternative (NI executed, not skipped) is not a jump, the action taken for it—load, add, etc.—may take more than a single English word to describe. Join the words with an underscore to keep them separate from the negative alternative, like this:

```
            TG,U        A5,MAXPAK+1   . MORE THAN MAX PACKIDS?
            L,U         A5,MAXPAK     . YES,SET_TO_MAXIMUM
```

19.12 Use Procs To Generate Repetitive Code

Every time you code a sequence more than once, think seriously about making that sequence into a proc. Refer to the chapter on procs (16) to see examples.

19.13 Pipelining and Performance

All of the supported 1100/2200 systems have a feature, called pipelining, that increases the effective speed of the CPU. The phases which make up the execution of a typical instruction can be overlapped to a certain extent. These phases are operations such as:

- Fetching an instruction word from memory.
- Testing the op code to determine the type of operation to be done.

- Transferring portions of the instruction word into internal registers in the arithmetic section of the CPU, even if the function code is not for an arithmetic instruction.
- Calculating an effective U.
- Moving A_a into the arithmetic section.
- Fetching the contents of U.
- And so on...

In some systems, there can be as many as 4 instructions in progress at once, in varying states of completion. While it may take just as long to execute a single instruction, overlapping execution of multiple instructions increases the rate at which instructions are completed. They "come out of the pipe" more frequently. But this increase in speed is dependent on the independence of the operations done by instructions in the stream. Look at this short sequence of instructions:

```
L,U         X5,(1,TABLE)  .
L           A2,WORD01,X5  .
S           A2,SGN070     .
```

Obviously, the store cannot be started until the load of A2 is completed, which in turn cannot start until the L,U is finished. The CPU detects these dependencies and halts the pipeline to let preceding instructions "catch up". These dependencies are called register conflicts. (Any kind of jump will also halt the pipeline.) Look at this sequence of code and identify the register conflicts:

```
L           A1,USERID,X9  .
S           A1,TABUSR,X6  .
L           A2,PROJID,X9  .
S           A2,TABPRJ,X6  .
L           A3,ACCTID,X9  .
S           A3,TABACT,X6  .
L           A4,SUPS,X9    .
AU,U        A4,5          .
S           A5,TABSUP,X6  .
```

Can you see that the pipeline will halt for each of the 3 load/store pairs, and for the final trio of instructions? If these instructions are executed straight through once or twice per execution, this order is still the best way to get the job done. On the other hand, if these operations are in the middle of a loop executed millions of times per execution, efficiency is more important, and it is worth spending time to squeeze the last few nanoseconds of speed out of the CPU. Here is a faster way to do the operations above:

```
L           A1,USERID,X9  .
L           A2,PROJID,X9  .
L           A3,ACCTID,X9  .
L           A4,SUPS,X9    .
S           A1,TABUSR,X6  .
S           A2,TABPRJ,X6  .
S           A3,TABACT,X6  .
AU,U        A4,5          .
S           A5,TABSUP,X6  .
```

The modification shown has the following effects:

- It is faster, possibly twice as fast.
- It has not changed the logic at all.
- It takes longer to code, and more registers. After all, we could have done the loads and stores with only 2 registers, not 5.
- It also makes the code less readable.

The last point is the reason that optimization of this sort is limited to sequences of code that must be as fast as possible. Note that some register conflicts are unavoidable; witness the AU,U and the store which follows.

The point is that you need not spend time identifying every register conflict which might occur in your code. In fact, you could ignore register conflicts for the rest of your career and probably get away with it. But if you see code that looks like the second version of the last example, think twice before criticizing the coder; he may well have known what he was doing.

19.14 What Goes First?

Here is a suggested order for the parts of your program source:

- Proc references, such as AXR$, ELT$, configuration procs, and so forth.
- $EQUates which determine buffer sizes—things that somebody might have to find fast if the program stops working.
- Proc definitions.
- Miscellaneous scalars (single-cell data items). In each case, a $RES for a table should be followed by the $EQUFs for the table fields.
- Error messages and packets for them.
- If there are many pages of data items, it is a good idea to group the items used for similar functions. Start each new group on a separate page (use a /. image).
- The mainline itself, which should be clearly labeled as such.

- The subroutines of the level called directly by the mainline.
- Subroutines called by the first top-level subroutine, then those called by the second, and so forth.
- The contingency handling path, if any.
- Other error paths, with their print ERs. Each should be clearly labeled with the condition that caused the jump there.
- Common or utility, bottom-level subroutines.

This list is provided as a guide. Deviate from it as necessary, but use some logical scheme. If there are so many subroutines that you yourself have trouble finding your way, put a list of the subroutine order and their descriptions at a strategic place in the code.

Chapter 20: Basic Data Structures

20.1 Begin With A Little Theory

Any program that deals with more than one of something must impose some sort of structure on the items. In many cases, a simple linear list, storing items in adjacent memory locations, using subscripts or a single index register, will do the trick. But often, we need to perform operations on the list that make a linear list difficult or expensive to use. In those situations, we need to apply just a little theoretical knowledge to solve the problem elegantly.

An analogy is interesting: a well-known programmer once compared programming to car repair. If you are working on your car at midnight, and you need a part, you can not finish without it. If you are programming at midnight, and you need a sort routine, you just create what you need out of your own creative intelligence. Knowledge of data structures will expand your stock of "spare parts".

If you want to go deeper into the theory of information structures, there are two excellent books: Fundamental Algorithms, Volume 1 of Donald E. Knuth's (3/5 complete) landmark series on "The Art of Computer Programming", and Fundamentals of Data Structures, by Horowitz and Sahni, ISBN 0-914894-20-X.

20.2 Definitions

A data structure is a set of data including structural relationships; an information structure consists of a data structure and the algorithms which use it. You are probably familiar with the simplest sort of data structure, the list. A list is an ordered collection of items. The items in a list, or in any data structure, are known as nodes. A node is a set of normally inseparable fields treated as a unit in a data structure.

This text uses the terms "node" and "record" interchangeably, but the word node implies the function and usage of related data in a data structure. The word record implies the use of related data in some sort of application program.

A data structure may be found inside another data structure. For example, the Program Control Table (PCT) is treated as a node in the PCT chain of OS/1100, but contains within it another whole set of structures: the Activity Save Area, name tables, file items, etc. In fact, half the job of understanding OS/1100 internals consists of learning the data structures in the product. The same thing applies to many other products.

To communicate algorithms, we must have a few notational conventions. X<-Y means we will give X the value of Y, just like the Fortran statement X=Y. The expression NEXT(THIS) means that the variable THIS points to a node, and that NEXT is a field within that node. So, to combine these constructs into SAVE<-PREV(THIS) means to give SAVE the value of the field PREV in the node which THIS points to.

Another definition: to traverse a list means to look at each of the items in a data structure in a predictable, repeatable way. In the subscripted table you already use in Fortran or Cobol, traversing it uses a simple loop with the loop counter/subscript beginning at one end of the list and incrementing by one for each entry in the table.

20.2.1 Linear and Linked Lists

The simple subscripted list or array you are familiar with is a linear list. Each entry has a fixed size, and there is no space between the nodes. Adding a node is simple, if you aren't concerned with the order; you just bump the pointer to the last node by the size of the nodes, and store your information in the node thus allocated.

If the nodes have to end up sorted, the problem is tougher. You can store them linearly as they come in, and sort them on output. That probably means you'll have to call an external (disk or tape) sort (a bit of overkill), or code a sort yourself, not a trivial matter if efficiency is important.

Another alternative is to maintain the sort order as the items are added to the table. But how do you insert a new item between other items in the list? You can move items which follow the insertion point "down", to higher subscripts, vacating a space for the new item. If the list gets very big and the entries are well out of order, the program could spend a lot of effort on just moving things around. Deletion is a similar problem, unless it's OK to mark the deleted items and repack the table when you run out of room.

There are variations on the linear list, such as using variable-sized nodes, but the limitations are similar.

The author's favorite data structure is the linked list. Now, there are times when it is not appropriate. If the only tool you have is a hammer, every problem you see looks like a nail. The trick is to recognize when a given tool is needed. But the linked list solves so many of the basic problems in keeping track of an ordered group of items, it's worth learning.

A linked list is a group of nodes allocated non-contiguously in memory. The way we impose order on them is by making each node point to the next in the list (see Figure 20-1). The last node normally points to nothing (that is, the link field contains a value of zero), or the first node in the list. The idea of "point to" can be either an actual program address, or a simple subscript (if you are using a high-level language), or something in between.

Linked lists can be found in different varieties: single-linked, double-linked, and circular. A single-linked list uses just one link field to point to the next item on the list. A double-linked list uses two link fields in each node; one points to the next node in the list, the other to the previous node. The backwards link is a big help when it's necessary to go backwards in the list.

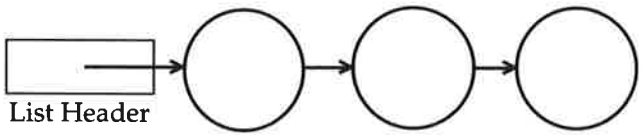

Figure 20-1. Single-Linked List

In a circular list, the last node points to the first. A circular list can be helpful, for example, if we have a number of mass storage units, and we need to rotate assignments among them. Or, if we want to assign students equally to a group of sections of the same class in a school, we would use a circular list of the class sections.

Obviously, a list of any kind doesn't exist by itself in space. We use a pointer called a list header to keep track of where the list begins, and may sometimes use a list trailer to point to the end of the list. (See Figure 20-2 on next page for a single-linked circular list with header and trailer pointers.)

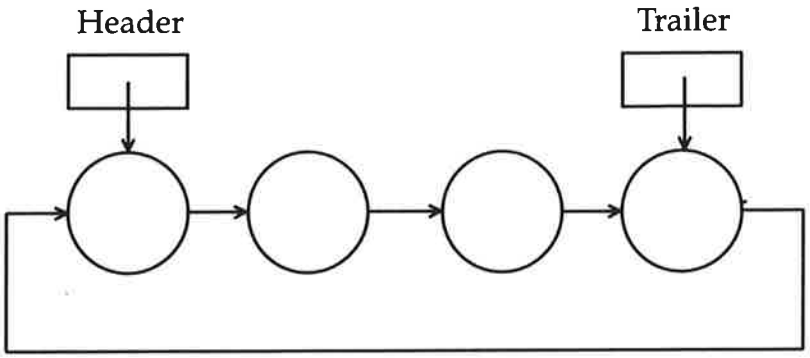

Figure 20-2. Circular Single-linked List

One problem sometimes encountered in programming is having more than one list. Each list may grow or shrink considerably, and it may be hard to predict by how much each may grow and in what manner the growth correlates with growth of others. For just two lists, the situation is simple. Two linear lists can exist in the same address space; one starts at the low-address end of the space, and grows toward higher addresses. The other starts at the high-address end, and grows toward lower addresses. (Hint; try using a negative increment in an index register.) Hopefully, they will not meet in the middle.

What about three or more lists? How can you place three lists in a single address space and allow them to grow and shrink in a fashion which maximizes the use of the space? It is impossible to do it simply, using linear lists. This is another situation in which linked lists shine. If we divide the available memory into equal-size nodes, and allocate one each time we need a new node, we can create as many lists as we like. The only additional overhead for each list is a new header for each list.

The price we pay for all this flexibility is overhead. A linear list requires very little overhead: just a pointer to the last item, or a count of the number of items. A linked list needs a list header, and each node contains one or more pointers to other nodes. For a list containing only a few hundred nodes, each of which contains a large amount of usable data, the overhead is reasonable. As the amount of data in each node goes down, or the number of pointer fields increases, the percentage of overhead goes up.

What have we bought with the overhead? We have preserved the structure inherent in the data. We have increased the efficiency of certain operations. Choosing the right data structure involves juggling the structure of the data, operations to be performed, the volume of data, and other factors.

20.3 MASM Code For Data Structures

In this part, some of the ideas discussed so far only in the abstract will be cast into code. And, we'll go into principles you can use to adapt the basic techniques to your application. If you don't follow the operations described here, or have trouble designing your own data structures and algorithms, don't hesitate to draw pictures. The strongest mind is no match for the weakest pencil!

20.3.1 Avail List

First, where do all those nodes come from when needed, and where do they go back to? Why, an avail list, that's where. ("Avail" is short for available, and it makes a much better MASM label.) An avail

list, depending on the application, can be a simple $RES statement in MASM (unstructured memory space), or a linked list built at assembly time.

It's a good idea to initialize the node space to zeroes at assembly time, but you may not like to see pages of zeroes in the listing. The ZRES proc from earlier is repeated here for your convenience:

```
ZRES*       $PROC
            +           0
            $UNLIST
            $DO         ZRES(1,1)-1 , + 0 .
            $LIST
            $END
```

The proc generates one word of zeroes before turning off the listing, to ensure that the proc reference line appears in the listing.

If the actual data list will be built repetitively at execution time, from scratch each time, just ZRES enough words to store the maximum number of nodes. The code which builds the list can insert the pointer fields:

```
NODECT      $EQU        200           . NUMBER OF NODES
NODESZ      $EQU        5             . NODE SIZE (WORDS)
NEXTND      $EQUF       0,,H2         . PTR TO NEXT NODE ON LIST
NODEBF      ZRES        NODECT*NODESZ . NODE BUFFER
```

Code examples to follow will assume the presence of equates for NEXTND, NODECT and NODESZ. NEXTND is the field within each node which will contain a pointer to the next node on the list.

If the application must create one or more lists which will grow and shrink dynamically, it is better to create the avail list as a linked list at assembly time.

```
AVAIL       +           NODEBF      . POINTER TO AVAIL LIST
NODEBF      .                       . NODE BUFFER
            $REPEAT     NODECT-1    . REPEAT FOR ALL BUT LAST
            +           0,$+NODESZ  . GENERATE "NEXT" POINTER
            ZRES        NODESZ-1    . ALLOC REST OF NODE
            $ENDR                   . END OF REPEATED NODES
. LAST NODE IN LIST:
            +           0,0         . LAST NODE HAS NULL LINK
            ZRES        NODESZ-1    . REST OF LAST NODE
```

The cell AVAIL will serve as the list header for the avail list. The $+NODESZ in each node puts a pointer to the next node into each node. The obvious exception is the last node, which has a zero for the pointer.

Alert readers will probably notice that there is one redundant line in this example. The + 0,0 for the last node in the list could easily be generated by ZRESing NODESZ words, not NODESZ-1. But creating the last node this way creates a zero next pointer explicitly, in the same way as in the other nodes. Keeping this kind of consistency takes only a bit longer to code, minimally longer to assemble, and no extra time whatever to execute.

20.3.2 Node Layout

MASM code should use one $EQUF for each field, rather than offset and j-factor in each field reference. A node in MASM might be described this way:

```
NODNXT    $EQUF     0,X5,H2      . NEXT LINK
NODKEY    $EQUF     1,X5,W       . KEY FIELD
NODNFO    $EQUF     2,X5,W       . NODE INFORMATION
NODESZ    $EQU      (NODNFO+1)**0777777  . NODE SIZE
```

The last line defines NODESZ based upon the offset for the last field in the node. ANDing the sum NODNFO+1 with 0777777 isolates the offset value from the $EQUF, removing the X register portion. Parameterizing the node size in this way ensures that as new fields are added to the node description, the buffer space expands to match. You can also "chain" $EQUFs together as shown in the last chapter (MASM style).

More on the subject of node fields: Sometimes, you may need to code a comparison of two nodes, for sorting purposes, or whatever. Putting the X register into the $EQUFs can get you into trouble when the code must look at two nodes at once. It is possible to grab the necessary key fields and compare them in registers, but that is messy. In this situation, add the following $EQUFs for just the key and NEXT fields:

```
XXXNXT    $EQUF     0,,H2        . NEXT FIELD, NO X REG
XXXKEY    $EQUF     1,,W         . KEY FIELD, NO X REG
```

Use of these $EQUFs should be limited to sort routines and the like. References to them must include the X register which points to the intended node.

The key field of the node can be trickier to design. If there is only one key, and it fits into 36 bits or less without modification, you're home free. If not, here are some suggestions. First, try scaling the number by a constant. If a comparison of two numbers can survive dividing both by 64 or other power of 2, and the lost precision won't be a problem, a simple shift will do the job. Or, you may have to compress a value using other means, perhaps folding a dozen disk equipment types into two bits according to speed, using a lookup table.

Second, you may very well have to put more than one field into the key, say a disk equipment type, an assign count, and availability, all in 36 bits. If one of those subkeys must be sorted in descending order, while the others are in ascending order, the subkey may have to be "inverted" — scaled, then subtracted from a constant. A multiple-key sort algorithm can get complicated and use a lot of overhead. Use your creativity to scale, cram, invert, or whatever is necessary to create a single, ascending 36-bit key field.

20.3.3 Traversing A Linked List

Eventually, we will probably have to look at all the nodes on a list, to find a candidate disk unit, a certain file, or whatever. The code on the following page does it:

```
          L,U       X6,0       . INIT PTR TO FOUND NODE
          L         X5,HEAD    . INIT NODE PTR
          L         A5,SENTNL  . SEARCH VALUE
P3N020    . NODE SEARCH LOOP
          TNZ,U     0,X5       . EMPTY LIST/NO NODE?
          J         P3N090     .  YES,P3N090  NO,
          TNE       A5,NODNFO  . THIS THE NODE WE WANT?
          J         P3N030     .  YES,P3N020  NO,
          L         X5,NODNXT  . GET ADDRESS OF NEXT NODE
          J         P3N020     . LOOK AT NEXT NODE
.
P3N030    . CANDIDATE FOUND
.         . . .
P3N090    . HIT END OF LIST, NO NODE QUALIFIES
          . . .
```

The TNZ,U is a quick way to test the modifier portion of an X register by itself. With the following jump, it's almost as easy to code as a JZ for an A register.

Really, the two load X5 instructions are the heart of the algorithm; the remainder is just window dressing. However, in any linked list algorithm, there is invariably a certain percentage of code devoted to boundary checks: testing for end of list, empty list, etc. A fair percentage of bugs in programs using linked lists can be traced to missing boundary checks.

20.3.4 Linked List Insertion

If a linked list is not built statically at assembly time, it is built at execution time by insertion, one or more nodes at a time. To insert a node, we need a pointer to the insertion point. Call this pointer CURRENT, and the new node NEW. (See Figure 20-3, following page.) Reduced to the minimum, we just set NEXT(NEW) to NEXT(CURRENT), and NEXT(CURRENT) to NEW (the dotted lines in Figure 20-3). (NEXT(NEW) means the NEXT field of the node at NEW.)

The MASM code to make this happen is:

```
          L         X4,NEW      . POINTER TO NODE TO INSERT
          L         X5,CURRENT  . POINTER TO INSERTION POINT
          L         A1,NODNXT   . GET CURRENT'S NEXT POINTER
          S         A1,XXXNXT,X4 . NEXT(NEW)<-NEXT(CURRENT)
          S         X4,NODNXT   . NEXT(CURRENT)<-NEW
```

Note that if CURRENT happens to be the last node in the list, its NEXT field will be zero. Setting NEXT(NEW) to NEXT(CURRENT) will have the effect of carrying the end-of-list sentinel — zero — to the new node at the end of the list.

20.3.5 Linked List Deletion

You can probably guess what deleting a node from a linked list is like. But if the list is single-linked, like these examples, one extra wrinkle is necessary. We must maintain a pointer which lags one behind the current node pointer as the list is traversed. Without that or backwards pointers (as in a

double-linked list), once we found the desired node, we would have to start over at the beginning of the list to find the node that points to the one to be deleted.

If we call the lagging pointer BACK1, removing CURRENT from the chain is simple: just set NEXT(BACK1) to NEXT(CURRENT). (See Figure 20-4, bottom figure this page.) Once again, if CURRENT was the last in the list, the end-of-list sentinel is maintained. Assuming that BACK1 is in X4, and CURRENT is in X5, here is the MASM code:

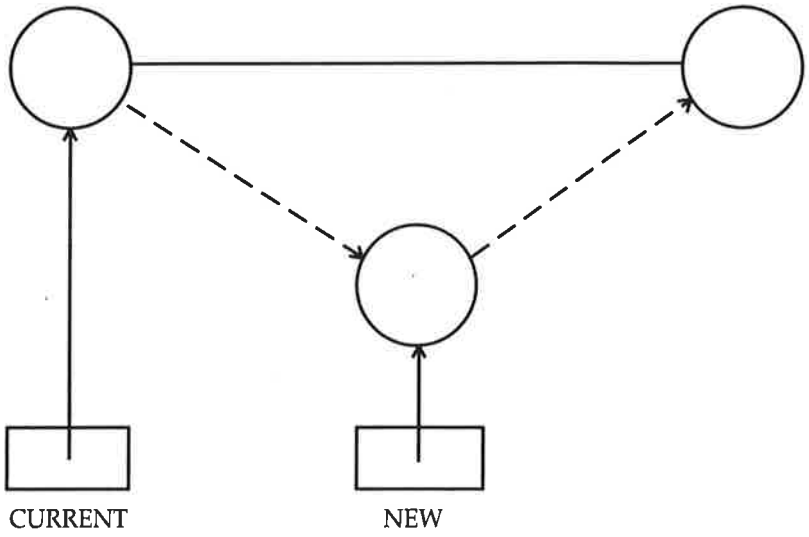

Figure 20-3. Linked List Insertion

```
L       A1,NODNXT       .
S       A1,XXXNXT,X4    .
```

20.3.6 Linked List Sorting

There are two types of requirements for sorted linked lists: One, we may build a list once, then sort it, followed by no further changes. Or two, the list might need to be maintained in sorted order as it is modified dynamically. Very similar code can be used for either requirement.

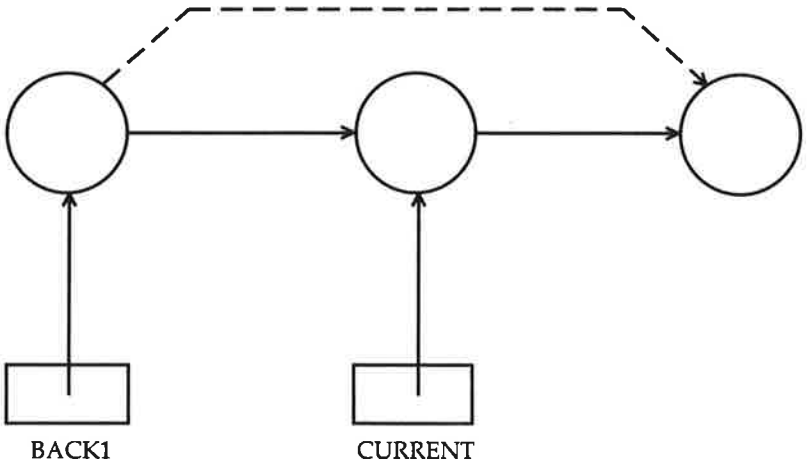

Figure 20-4. Linked List Deletion

In the first case, almost any type of sort will do: bubble sort, exchange sort, or what-have-you. Particularly suited to linked lists is the insertion sort. To do an insertion sort, detach all the nodes from the list header, and add them to another list. (Note that this is very simple: set HEAD2 to HEAD, then HEAD to 0.) Then pass each node to a routine that places it back on the original list (starting with HEAD), but in sorted order. Begin pulling off nodes beginning with HEAD2.

The second case, maintaining sorted order, needs the same logic as the inside of the insertion sort.

Here is one example of an insertion sort. There are two parts, both of which traverse lists. The first part traverses the unsorted list, pulling off nodes which are passed to the second part, a subroutine. The subroutine performs the insertion, locating the insertion point and inserting the new node there. The code takes the nodes from the list whose beginning is in HEAD1, and places the start of the new list in HEAD2. The code uses the following pointers:

- X6 is the pointer to the node to be inserted (NEW).
- X4 points to the node before the trial insertion point (BACK1).
- X5 points to the node after the trial insertion point (CURRENT).

```
HEAD1     +          $-$        . (ASSUME SET AT XQT TIME
HEAD2     +          0          . HEAD OF SORTED LIST
NODNXT    $EQUF      0,X6,H2    . NEXT POINTER IN NODE
XXXKEY    $EQUF      1,,W       . KEY FIELD OF NODE
.                               . (NEEDS X REG)
XXXNXT    $EQUF      0,,H2      . NEXT POINTER (NEEDS X REG)
. INSERTION SORT DRIVER:
          L          X6,HEAD1   . START ON FIRST LIST
          SZ         HEAD2      . INIT START OF NEW LIST
PH6010    .
          L          A7,NODNXT  . SAVE POINTER TO NEXT
          LMJ        X2,P6SORT  . INSERT THIS NODE IN NEW LIST
          L          X6,A7      . GET PTR TO NEXT NODE
          JNZ        A7,PH6010  .  MORE_NODES,PH6010
          L          A0,HEAD2   . HEAD1<-HEAD2,
          S          A0,HEAD1   . AND SORT IS DONE.
.
.         (END OF INSERTION SORT DRIVER PORTION)
.
P6SORT    . INSERTION SORT SUBROUTINE. X6 POINTS TO NEW NODE.
          SZ         NODNXT     . CLEAR NEXT POINTER
          TNZ        HEAD2      . OUTPUT LIST EMPTY?
          J          P6S050     .  YES,P6S050  NO,
          L          A3,XXXKEY,X6 . GET KEY OF NEW NODE
          L          X5,HEAD2   . INITIALIZE CURRENT POINTER
          SZ         X4         . INITIALIZE BACK1 POINTER
P6S010    . COMPARE KEY(CURRENT) WITH KEY(NEW)
          TG         A3,XXXKEY,X5 . KEY(CURRENT) > KEY(NEW)?
          J          P6S040     .  NO,P6S040  YES,
P6S020    . KEY(NEW) < KEY(CURRENT). INSERT NEW NODE BETWEEN
          . NODES AT X4 AND X5 .  BACK1 MAY BE ZERO.
          S          X5,XXXNXT,X6 . NEXT(NEW)<-CURRENT
          TNZ        X4         . BACK1=0?
          J          P6S050     .  YES,INSERT_AT_BEGINNING
P6S030    .
          S          X6,XXXNXT,X4 . NO,NEXT(BACK1)<-NEW
          J          P6S080     .
.
P6S040    . STEP X4, X5 TO NEXT NODES, IF ANY.
          L          X4,X5      . BACK1<-CURRENT
          L          X5,XXXNXT,X5 . CURRENT<-NEXT(CURRENT)
          TZ         X5         . NEW CURRENT=0?
          J          P6S010     . NO,TRY_NEXT_NODE_IN_LIST
          J          P6S030     . CHAIN END REACHED, ADD THERE
.
P6S050    . INSERT NODE AT BEGINNING OF LIST.
          S          X6,HEAD2   .
P6S080    . ALL DONE WITH THE NEW NODE.
          J          0,X2       .
```

The key to inserting a node in a sorted list is to traverse the list, maintaining both a CURRENT pointer (X5) and a BACK1 pointer (X4). When we find that KEY(CURRENT) is greater than KEY(NEW), we insert the new node between BACK1 and CURRENT. If BACK1 is zero (TNZ X4), CURRENT is pointing to the first node in the list, and we must insert NEW at the head of the list.

20.4 Variations In Linked Lists

Depending on the way we choose the insertion and deletion points in a list, it can implement a stack or a queue. You've heard of stacks; just like the add-on-the-top, remove-from-the-top devices in your local cafeteria, the first node added to a stack is the last removed, and vice versa. A stack needs only one list pointer, and it points to the top of the stack.

To add a node to a stack (and assuming NEW points to the node to be added), set NEXT(NEW) to HEAD, and HEAD to NEW. To remove a node (and leave NEW pointing to it), set NEW to HEAD, and HEAD to NEXT(HEAD). (The latter assumes that HEAD is non-zero. If HEAD is zero, we have stack underflow, generally a non-recoverable error condition. Don't forget to check for this kind of error. You will find that perhaps one third of your code goes to checking for "boundary conditions" like stack underflow.)

Queues are like the lines for bank tellers (the old way, not the single-line-for-all-tellers in use today); the first node on the list is the first taken off, etc. A queue needs a pair of list pointers (see Figure 20-5). HEAD actually points to what we thought of earlier as the end of the list, and TAIL what was the start of the list. Nodes are added at the HEAD end, and removed from the TAIL end.

To add a node to a queue, set NEXT(HEAD) to NEW (if HEAD is non-zero), and set HEAD to NEW. To take a node from the queue, set TAIL to NEXT(TAIL) (assuming TAIL is non-zero). Once again, we spend about a third of our coding effort to cover the boundary condition (namely, list empty).

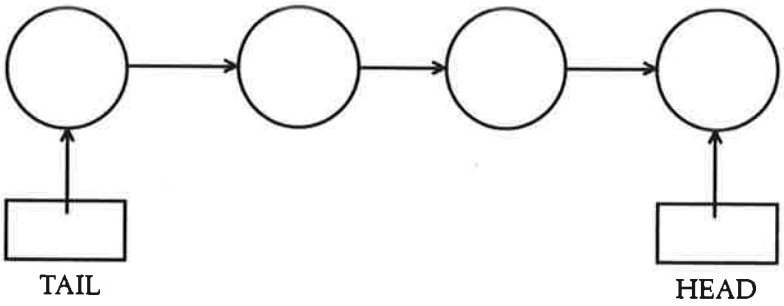

Figure 20-5. Queue

For debugging purposes, we may want to change the behavior of an avail list. That's because if we bother to put back nodes which we no longer need, the avail list will behave like a stack. A good portion of the recent history of the program is erased. If we simply maintain two pointers to the avail list, and add at one end, and take from the other, we now have a queue. The history of the program is maintained, at least to the extent that you allocate extra nodes.

Putting both forward and backward links in each node changes the rules somewhat, creating a double-linked list. Insertion needs only a couple of extra stores to maintain backward pointers. Deletion and sorting won't need that BACK1 pointer to assist in locating the node whose forward pointer must be changed. But most of all, maintaining both pointers allows traversing the list in either direction, even starting in the middle if such a pointer is available.

20.5 Summary

Linked lists give you a powerful new tool for keeping track of files, disk units, userids, and so forth. Give them a try in your next MASM program.

Exercises

E20-1. (Project) Write a program to read in a list of names in ASCII. Sort them into alphabetical order by the first 4 characters. You may sort them as you read them, or read all of them and sort just before printing. Use a linked list.

E20-2. (Project) Write a program to read in a list of names and Social Security numbers. Print them out in order by SSN. (Hint: in order to print and sort by the SSN, you may need to preserve it in both ASCII and binary forms.) Again, use whatever sort strategy you wish, but use a linked list.

Chapter 21: Using Character Data

21.1 Basics of Character Manipulation

As you explore the tools and techniques of MASM programming, it's hard to avoid dealing with character manipulation. Since the 1100 and 2200 systems are word-oriented, the way to move a character at a time may not be immediately obvious.

Recall that the 36-bit 1100 data word can be divided into characters two different ways: first, as six 6-bit characters, normally using the Fieldata character set; second, as four 9-bit characters, normally using the ASCII character set . ASCII stands for American Standard Code for Information Interchange, and it is now the major industry standard outside of IBM's Extended Binary Coded Decimal Interchange Code — EBCDIC.

Quarter-word ASCII, unlike Fieldata characters, are stored with a bit of slack in each partial word. Since ASCII is only an 8-bit code, the leftmost bit of each quarter word is unused. In this case, the loss of a small amount of overhead space is more than made up for by the increase in processing speed.

Before we go moving characters around with wild abandon, the CPU must be set up to cooperate with our efforts. The CPU on each 1100 system has a mode setting which affects some partial word transfers. The j values 4 to 7 have different effects according to whether the CPU is in quarter- or third-word mode (see Chapter 7). If the quarter word mode bit in the CPU/IP's designator register is set, j values of 4 through 7 correspond to quarter words; if clear, 4 through 7 correspond to the third words plus XH1.

The beginner may protest, "But I coded Q1 on my instruction! Why didn't I get a quarter word?". Remember that Q1 is just a label corresponding to a number at assembly time. The interpretation made by the CPU at execution time is dependent on the setting of the designator register's quarter-word mode bit, regardless of the programmer's intentions at assembly time.

There are 5 ways to control quarter-word mode:

- The F option on the @MAP processor call requests quarter-word mode. The T option requests third-word mode.
- Also at collection time, you can use a TYPE THIRD or a TYPE QUARTER directive in your map source images.
- Put a '$INFO 1 3' directive in your MASM source to request quarter-word mode from the start of execution.
- If the program has to switch back and forth between quarter-word and third-word mode, the least CPU-efficient is via ER PSR$. PSR$ requires a 36-bit parameter in A0. For each possible mode setting in the designator register (that we are allowed to change), there are two bits in the A0 parameter mask. One bit controls whether the mode is to be changed, and the other gives the new value. The code to set quarter-word mode is:

```
L       A0,(0400001)  .
ER      PSR$          .
```

or:

```
L,U         A0,0400001  .
ER          PSR$        .
```

To return to third-word mode, change the parameter going into A0 to 01. PSR$'s operational code is just a few instructions, but the program must still pay for the price of ER handling. (After all, an Executive Request instruction still causes an interrupt, and each interrupt takes time to process.)

- The most efficient way to control quarter-word mode is via the LPD and SPD instructions — Load/Store Processor Designator. These instructions allow the programmer to change the hardware directly. You may not be concerned with the settings of double-precision underflow mode, floating-point compatibility mode, or arithmetic exception interrupt control. If you are not, you can simply do an "LPD 010" to set quarter-word mode. If you do not want to affect the settings of other modes, the PRM suggests code similar to this (UP-4144.32, Section 4.10.2.2, UP-4144.38, Section 23.5):

```
SPD         A2          . GET OLD DESIG BITS
OR,U        A2,010      . OR IN QUARTER WORD
LPD         0,A3        . PUT BITS BACK
```

  Clearing quarter-word mode works similarly:

```
SPD         A2          . GET OLD DESIG BITS
AND,U       A2,-010     . AND OFF QUARTER WORD
LPD         0,A3        . PUT BITS BACK
```

In each case, the SPD acquires the current settings of the mode bits, and the LPD returns new settings. The OR adds the bit to be set, and the AND clears the desired mode bit. (There are also designator bits for character addressing mode and active BDR pair, so if the MASM code is to be called by Cobol or used in a program which takes advantage of the utility PSR, the three-instruction sequences are strongly recommended.)

Note that sixth-word moves (as in Fieldata) are not affected by the setting of quarter-word mode.

21.2 Character Vectors

Once the CPU mode setting is taken care of, you can go on to the main bit of magic it takes to move characters easily, the character vector. Imagine how you would solve the problem if you started from scratch, without knowing about character vectors. You would need two things to keep track of where you were working in a buffer full of characters:

- A word index ("which word are you looking at right now?") and
- A character index ("which character within that word?").

The word index is easy to use; the 1100 hardware handles that directly. The character index is more difficult, since the 1100 can't index directly to the character level. The trick is to make the character index point to an instruction which loads or stores that portion of a word.

Without further ado, here is a character vector for Fieldata:

```
GETFDV      L,S1        A3,0,X5
            L,S2        A3,0,X5
            L,S3        A3,0,X5
            L,S4        A3,0,X5
            L,S5        A3,0,X5
            LMJ         A2,$+1
            L,S6        A3,0,*X5
            LXM,U       X6,0
            J           0,A2
```

Can you guess which index register is the word index and which is the character index? The X5 appearing on most of the instructions should be a clue to its use as the index to the target word. X5 gets bumped whenever we get to S6 and must step to the next word. That leaves X6 as the character index. But we are getting ahead of ourselves. How do we use this list of instructions?

The best way to call on a character vector is via an EX instruction in a proc, named mnemonically:

```
GETFD*      $PROC
*           EX          GETFDV,*X6
            $END
```

We now have only two things missing: the setup, and the use of the data. Here is the setup to start at the beginning of the buffer CHAR01:

```
            L           X5,(1,CHAR01)
            L           X6,(1,0)
```

The code to use the data is simple, too:

```
            L           A0,CHARLEN
            AN,U        A0,1
            S           A0,A1.
LOOP1.
            GETFD
            ... (Use character in A3)
            JGD         A1,LOOP1
```

How does this character vector work? For the first five passes through the loop, the first five instructions of the character vector are executed in sequence. That is, on each EX instruction, X6, the character index, is bumped to point to the next instruction in the character vector.

On the sixth pass, though, something different happens. The EX instruction "sees" an LMJ instruction. The LMJ when executed by an EX behaves differently from normal; the program address captured by the LMJ is not the location of the LMJ, but the location of the EX instruction. This allows us to do the J 0,A2 to return to the location following the EX instruction.

Why break the sequence in the first place? The sequence that must happen for the last character in a word takes more than one instruction. The first step gets the last character in the word; the other

step resets the character index to the beginning of a word. Notice the *X5 on the L,S6 instruction; this bumps X5 to point to the next word.

The procs and character vector use A2, A3, X5, and X6, leaving A0 and A1 free for ER calls.

A character vector to store, rather than load, characters is almost identical to the one we have just seen:

```
PUTFDV      S,S1        A3,0,X3
            S,S2        A3,0,X3
            S,S3        A3,0,X3
            S,S4        A3,0,X3
            S,S5        A3,0,X3
            LMJ         A2,$+1
            S,S6        A3,0,*X3
            LXM,U       X6,0
            J           0,A2
```

There are only two differences: first, each load instruction becomes a store; second, the index registers are different from those used with the load vector. (The latter assumes that both vectors will be used in the same section of code; we simply don't want the loads interfering with the stores.) The proc for storing Fieldata characters is identical to the "get" proc, except for the name of the store vector:

```
PUTFD*      $PROC
*           EX          PUTFDV,*X4
            $END
```

If the procs are created carefully, your code should be able to get, put, get, put without any excess verbiage except stopping when the source buffer is exhausted.

A character vector for use with ASCII is quite similar to that for Fieldata:

```
GETQWV      L,Q1        A3,0,X5
            L,Q2        A3,0,X5
            L,Q3        A3,0,X5
            LMJ         A2,$+1
            L,Q4        A3,0,*X5
            LXM,U       X6,0
            J           0,A2
```

This vector's behavior is more of the same: X6 is set to 0 to start, then gets values of 1, 2, 3, and then 0 again. X5 marches a little more slowly across the data buffer than X6: 0, 0, 0, 0, 1, 1, 1, 1, etc.

Next, let's add a couple of ideas to make the GETxxx procs more useful still. First, if we set up R15 with the number of characters in the buffer, this proc will jump to a desired address when all the characters have been pulled out:

```
P           PROC
GETQWL*     NAME
*           EX          GETQWV,*X6  .
            DEC         R15         .
            J           P(1,1),P(1,2)  .
            END                     .
```

It is easy to use this proc:

```
            L           R15,CHARLEN  .
LOOP2.
            GETQWL      0,X11.  (No More)
            ... (Use character in R3) .
            J           LOOP2.
```

The proc will jump to 0,X11 when the character count is exhausted.

(A final note about the procs: The * on the first instruction in each is a waiting label. See Chapter 16 for more about waiting labels.)

21.3 Changing Character Sets

We can also incorporate translation into a GETxxx proc. Using the SYSLIB element, TABLE$, we can go from Fieldata to ASCII or vice versa. This element is a 128-word table with the entry point ASCFDASC$. The first half of each word is the Fieldata equivalent of the ASCII character which indexes to that entry. (In other words, at an offset of 040 into the table, an ASCII space, we will find in H1 a 05, a Fieldata space.) H2 of each word is the ASCII equivalent of the Fieldata offset. (At an offset of 05 into the table, H2 is a 040.) A proc which would combine the Fieldata load and translate to ASCII would look like this:

```
GETFDA*     $PROC                   .
*           EX          GETFDV,*X6  .
            L,H2        A3,ASCFDASC$,A3  .
            $END                    .
```

And the ASCII to FD counterpart would be:

```
GETAFD*     $PROC                   .
*           EX          GETQWV,*X6  .
            L,H1        A3,ASCFDASC$,A3  .
            $END                    .
```

SYSLIB also defines a pair of translation routines, FDASC$ and ASCFD$. You can check these out in the SYSLIB manual, UP-8728.3.

21.4 Hardware-Dependent Solutions

The character-handling techniques just described will work on any supported 1100 or 2200 system. If your code must test each individual character as it scans the input buffer, the above generic code